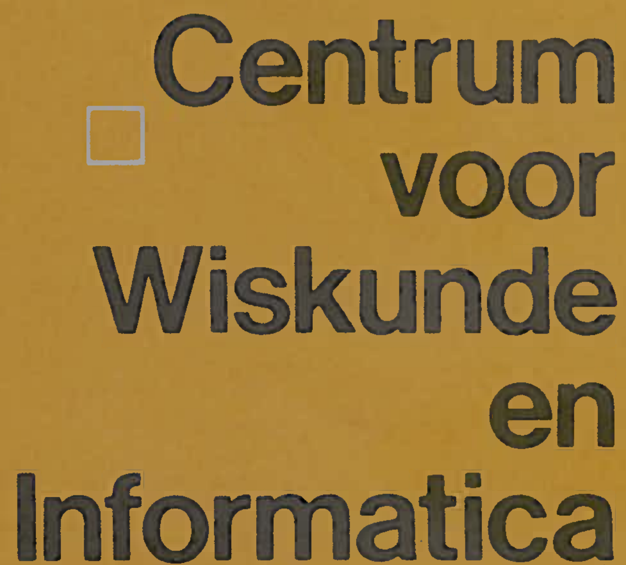


Centre for Mathematics and Computer Science

J.G. Rekers

Modular parser generation

Computer Science/Department of Software Technology Report CS-R8933 September

1989

Erratum

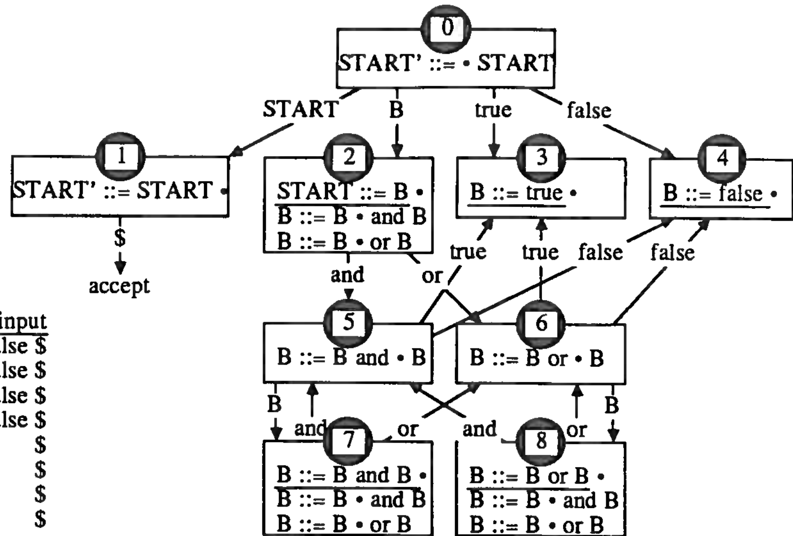
This figure should have been at page 5 of CWI report no. CS-R8933: *Modular Parser Generation* by *J.G. Rekers*.

BOOL ::= true
BOOL ::= false
BOOL ::= BOOL or BOOL
BOOL ::= BOOL and BOOL
START ::= BOOL

Grammar of the Booleans

Parse stack	input
0	true or false $
0 true 3	or false $
0 BOOL 2	or false $
0 BOOL 2 or 6	false $
0 BOOL 2 or 6 false 4	$
0 BOOL 2 or 6 BOOL 8	$
0 BOOL 2	$
0 START 1	$

Steps of the parser on input "true or false"



Graph of itemsets for the Booleans

Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

J.G. Rekers

Modular parser generation

Computer Science/Department of Software Technology Report CS-R8933 September

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Research (N.W.O.).

Copyright © Stichting Mathematisch Centrum, Amsterdam

Modular Parser Generation

J.G. Rekers
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
email: rekers@cwi.nl

We consider a parser generator MPG for grammars having a modular structure. Each grammar module contains a partial or incomplete grammar, which has to be supplemented with the grammars defined by the modules imported in it. MPG generates a parser for each module by generating a parser for the union of a given set of modules in such a way that parsers for the individual modules can be selected from it. This yields a separate parser for each module which is as efficient as a conventionally generated one, while the generation work invested in it can be re-used for other modules. The generator works fully incrementally, as a modification of the grammar is propagated to all parsers whose generation depends on it.

Key Words & Phrases: modular grammar definition, modular syntax composition, modular parser generation, multiple parser generation, incremental parser generation, LR parser generation.

1985 Mathematics Subject Classification: 69D12 [**Programming techniques**]: Automatic programming; 69D22 [**Software engineering**]: Tools and techniques; 69D26 [**Software engineering**]: Programming environments; 69D41 [**Programming languages**]: Formal definitions and theory; 69D44 [**Programming languages**]: Processors.

1987 CR Categories: D.2.2, D.2.6, D.3.1, D.3.4.

Note: Partial support received from the European Communities under ESPRIT project 348 (Generation of Interactive Programming Environments - GIPE).

1. Introduction

Modularization is a well-known method in software engineering to split a large program or specification into smaller components. Each module can to a large extend be developed and tested separately, and, when correct, it can be combined with other modules. Modularization encourages clearly structured solutions, and is beneficial to the re-usability of components.

1.1. Modular syntax definition

As grammars can grow quite large, and parts often can be borrowed from other grammars, modularization could be beneficial to the development of grammars as well. In view of the foregoing, we propose a system for the interactive development of modular syntax definitions:

- As the syntax defined by a module will in general be used by many other modules, and as we want to make an interactive system, the time needed to update each parser affected by a modification should be proportional to the size of the modification and not to the size of the grammar. We thus need a fast and *incremental* parser generator.
- The intended system allows a module to express its semantics in terms of the syntax introduced in the module itself and the modules imported by it. This means that not only a parser for the

Report CS-R8933
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

'top-module' of the definition must be generated, but that a separate parser must be generated for each module of the definition. We thus need a *modular* parser generator.

- We do not want to restrict the modular composition capabilities of the system by requiring that all rules for a non-terminal should appear in a single module, or limit the defined grammar to a certain class, or force the import relation to have a tree structure. Although these restrictions would simplify the implementation, they limit the range of possible modular definitions in an unacceptable manner.

We developed a parser generation method that makes it possible to distinguish the parser for each sub-module in the parser generated for the top-module. This means that during generation, not one but many parsers are generated, while double generation work is avoided. If one module is used in several contexts, only a single (part of the) parser is generated for it. We do not have to restrict the modular composition capabilities to be able to do this.

Even if a modular syntax definition does not have a single top-module, we still let the parser generator work on the union of all modules in the definition. It is true that this union defines a language that is too large, but as we use a *lazy* generation scheme, we will never generate more than is actually needed for the parsers of the modules in the specification. In addition to this, the generator is *incremental*, which means that a deletion or addition of a grammar rule in one module is propagated to all parsers that depend on it.

In summary, the Modular Parser Generator *MPG* has the following properties:

- It generates a separately usable parser for each module.
- No double generation work is done.
- When a module is modified, *MPG* updates all parsers that depend on the modified module incrementally.
- *MPG* generates efficient parsers and allows general context-free grammars.

MPG is based on the lazy and incremental parser generation techniques introduced in [HKR89]. This method is in turn based on LR parse table generation and parallel parsing.

1.2. Related work

The parsing system CIGALE [Voi86] allows interactive modification of grammars and import of previously defined parsers. It is not entirely clear which class of grammars is accepted by the system. It solves the problem of ambiguous parses by returning the "first" parse of a sentence. The parse time of CIGALE is acceptable only for small input sentences, as the parser works in exponential time.

OBJ2 [FGJM85] also allows composition of modules defining their own syntax. It uses a recursive descent parser with backtracking, but, as far as we know, the modular aspects of this parser have not been described yet.

2. A formal definition of Modular Parser Generation

We will first define what (partial) grammars, parsers and modules are. Next, without going into detail, the properties of *MPG* are given and our solution for modular parser generation is sketched.

2.1. Definitions

A grammar is a tuple $<V_T, V_N, R, \text{START}>$, with terminals V_T, non-terminals V_N, $V_T \cap V_N = \emptyset$, start-symbol $\text{START} \notin V_T \cup V_N$, and rules $R \subset \{V_N \cup \text{START}\} \times (V_T \cup V_N)^*$. A rule is normally written as $A ::= \alpha$.

The incremental parser generator [HKR89] consists of the following functions: $IPG(G)$ generates a parser for grammar G, $ADD(P, r)$ extends parser P with rule r (if r is correctly typed), and $DELETE(P, r)$ removes rule r from parser P (if r was part of P).

A modular syntax definition consists of a set of modules $\{M_1, \ldots, M_n\}$. Each module M_i is a triple $<N, I, G>$ consisting of the name of the module N, a set of names of imported modules I,

and a *partial* grammar G. G need not be complete because rules in G may use terminals and non-terminals of the grammars imported in the module.

A modular syntax definition S is called well-formed if (1) all imported modules are defined in it, (2) the corresponding import graph is a-cyclic, (3) the partial grammars of all modules use the same start-symbol, and (4) the full grammar $grammar(M, S)$ of each module M in definition S is a correct grammar, with

$$grammar(<N, \emptyset, G>, S) = G$$
$$grammar(<N, \{N_1, \ldots, N_n\}, G>, S) =$$
$$G \cup grammar(<N_1, I_1, G_1>, S) \cup \cdots \cup grammar(<N_n, I_n, G_n>, S).$$

The union of grammars is defined as

$$<V_T, V_N, R, \text{START}> \cup <V_T', V_N', R', \text{START}> = <V_T \cup V_T', V_N \cup V_N', R \cup R', \text{START}>$$

when $(V_T \cup V_T') \cap (V_N \cup V_N') = \emptyset$.

2.2. The Problem

The modular parser generator *MPG* has to generate a parser for module M in modular syntax definition S. What are the possible ways to implement *MPG*?

We could of course use *IPG* (Section 2.1) to generate a parser P_i for each module M_i in the specification: $P_i = IPG(grammar(M_i, S))$. To implement *MPG* in this way would be quite a waste, because each module defines a separate language for which a separate parser has to be generated. We can thus assume that for each module in the imports of M_i, a parser has already been (or will have to be) computed.

An intuitively better way would be to use *IPG* only to generate parsers for modules without imports. The parser for a module with imports should then be composed of the parsers of the imported modules plus its own syntax. However, composing two parsers P_{G_1} and P_{G_2} is non-trivial; the grammars G_1 and G_2 can influence each other, and P_{G_1} and P_{G_2} have to be merged in a more or less intricate way if a correct parser $P_{G_1 \cup G_2}$ is to be obtained.

IPG uses global information about its input grammar. It has to know all rules that derive a certain non-terminal and takes together rules with the same prefix. Because, in general, this information in $G_1 \cup G_2$ will differ from that in G_1 or G_2, $P_{G_1 \cup G_2}$ will not have much in common with P_{G_1} or P_{G_2}. So, with our parser generation technique, constructing the union of two parsers is difficult, and will in most cases just consist in generating $P_{G_1 \cup G_2}$ by applying *IPG* to $G_1 \cup G_2$.

2.3. The Solution

As it is difficult to combine two parsers without repeating the whole generation process, we go the other way around. We only generate the combined parser and extract smaller parsers from it as is shown in Fig. 1.b.

Fig. 1.a: Construct the union of parsers, **1.b**: Restrict a parser

We do not generate P_{G_1} and P_{G_2}, but only generate $P_{G_1 \cup G_2}$. This parser can also be used as P_{G_1}, by only allowing it to generate trees according to G_1. Hence,

$$P_{G_1} = G_1 \square P_{G_1 \cup G_2}$$
$$P_{G_2} = G_2 \square P_{G_1 \cup G_2}$$

where $G \square P$ is something like "observing P through a window G" or "restricting P to G". This restriction yields exactly the desired parser, as the conditions for P to be a parser for G are:

- P should recognize all sentences in the language defined by G.
  The sentences recognized by $P_{G_1 \cup G_2}$ are a superset of the sentences of G_1.
- P may only generate parse-trees based on rules of G (or: reduce according to rules in G).
  This is just the restriction we impose on $P_{G_1 \cup G_2}$ with the $\Box$-operator.

Now we can implement $MPG(M_i, S)$ in the following manner: when $S = \{<N_1, I_1, G_1>, \ldots, <N_n, I_n, G_n>\}$, we generate a parser $P_{union} = IPG(G_1 \cup \cdots \cup G_n)$. Each parser P_i can now be obtained by restricting P_{union} to parse only according to the grammar of its module M_i: $P_i = grammar(M_i, S) \Box P_{union}$.

3. Incremental parser generation (*IPG*)

The implementation of the modular parser generator is based on the lazy and incremental parser generator *IPG* described in [HKR89]. We refer to that article for a complete description. In this paper we will only give the main algorithms in simplified form.

IPG consists of a parallel parser driven by incrementally generated LR(0) parse tables. The parallel parsing technique used was originally developed by Tomita and is described in [Tom85]. As this parser tries all possible parses in parallel, it is able to comply with arbitrary context-free grammars. The algorithm is quite efficient, as it joins parallel parsers whenever possible.

We will first describe the parser, the parse tables, and the ordinary LR(0) parser generation technique. This LR(0) generator will then, via a lazy generation technique, be extended to the incremental parser generation technique of *IPG*. A good introduction to LR parsing in general can be found in [ASU86, ch. 4.7].

3.1. The parser

To give an idea of how our parser works, we present a non-parallel parser. It maintains a parse stack, which initially contains the start-state, and repeatedly asks its parse table for actions to be performed in the current state on top of the stack and with the current input symbol. If there are several possible actions, it chooses one of them. A parallel parser would in this situation split up in multiple parsers, one for each possibility.

```
LR-PARSE(start-state, sentence):
  Push the start-state on the parse-stack
  while true do
    The current state of the parser is the state on top of stack
    The current input symbol is the first symbol of sentence
    Choose an arbitrary action from those returned by ACTION
    if there is no action then
      Reject the sentence
    else
      if it is a shift action to a state then
        Push the current symbol and that state on the stack
        Remove the current symbol from the head of sentence
      elseif it is a reduce action of rule A ::= β then
        Replace β on the stack by A
        Call GOTO for a new state
        Push that state on the stack
      elseif it is an accept action then
        Accept the sentence
      fi
    fi
  od
```

3.2. The parse table or graph of itemsets

The parser asks information from its parse table using the routines *ACTION* and *GOTO*. In fact, the parse table is a graph of itemsets. The nodes of the graph are itemsets and the (labeled) edges of the graph are formed by the transitions between itemsets. A state of the parser is an itemset in this graph (an example of the graph generated for a small grammar is given in Fig. 2). The parser moves through this graph: shift actions cause the parser to move forward along a transition labeled with the current input symbol, while reduce actions first cause a move backward along the path stored on the parse stack, and then a move forward along a transition labeled with the non-terminal that was the result of the reduction.

ACTION(*state, symbol*):
  result := {reduce A ::= β | A ::= $\beta \in$ *state.reductions*} $\cup$
    {shift *state′* | (*symbol state′*) $\in$ *state.transitions*} $\cup$
    {accept | (*symbol* accept) $\in$ *state.transitions*}
  return *result*

GOTO(*state, symbol*):
  return *state′*: (*symbol state′*) $\in$ *state.transitions*

Each itemset in the graph contains the fields *kernel, transitions, reductions* and *type*. The *kernel* of an itemset is a set of dotted rules that can be recognized by the parser in this state/itemset; the dot in such a rule indicates how far the parser has progressed in recognizing it. The *transitions* of an itemset are the labeled edges of the graph from the itemset to other itemsets. The *reductions* of an itemset are the rules that can be recognized completely by the parser in this state/itemset. The *type* of an itemset can be *initial* or *complete*. When *initial* the transitions and reductions have not yet been computed.

Fig. 2: An example

3.3. Parse table generation

The graph of itemsets is generated by an LR(0) parse table generator like the following routine *PG*:

PG(*Grammar*):
  Grammar := *Grammar* $\cup$ {START′ ::= START}
  Generate a start-itemset with as kernel {START′ ::= •START}
  while there is an initial itemset **do**
    Complete it using *EXPAND*
  od

return the start-itemset

The real generation work is done by *EXPAND*, which computes the transition and reduction fields of an itemset. It starts by using *K-CLOSURE* to generate a new set *Closure* of dotted rules (which is an extension of the kernel) containing all rules that may become applicable in this state/itemset. *Closure* is then partitioned in subsets of rules having the same symbol *S* after the dot. On shifting *S* (or reducing to *S*), the parser will have advanced one step recognizing a rule in the subset associated with *S*. For each *S* the associated subset is transformed into a new kernel by moving the dot over the *S*. When an itemset with that kernel does not yet exist, it is generated as an initial one. A transition, that indicates to go to that itemset when *S* has been recognized, is added to *transitions*, A rule in the extended kernel having a dot at the end has been recognized completely. It depends on the left-hand side of the rule whether this implies an accept or a reduce action.

```
K-CLOSURE(kernel):
  Closure := kernel
  while there is a rule in Closure with its dot before an S do
     Extend Closure with all rules that derive S
  od
  return Closure

EXPAND(itemset):
  Closure ::= K-CLOSURE(kernel)
  for each symbol S in Closure that is immediately preceded by a dot do
    Generate a new kernel with the dot moved over that S
    if there does not exist an itemset yet with that kernel then
      Generate an initial itemset with that kernel
    fi
    Add to transitions a shift action under S to that itemset
  od
  for each rule in Closure with the dot at the end do
    if it is START' ::= START• then
       Add an accept action to transitions
    else
       Add a reduce action of that rule to reductions
    fi
  od
  The itemset is now "complete"
```

3.4. Lazy parse table generation

The above parser generator can be made lazy by moving the completion of itemsets from the generation phase to the parsing phase. The parser generator *LPG* just generates the start-itemset as initial itemset, and routine *ACTION* tests if the incoming state/itemset is still initial; if so *EXPAND* is called first.

```
LPG(Grammar):
  Grammar := Grammar ∪ {START' ::= START}
  Generate a start-itemset with as kernel {START' ::= •START}
  return the start-itemset

ACTION(state, symbol):
  if state is an initial itemset then EXPAND(state) fi
  result := {reduce A ::= β | A ::= β ∈ state.reductions} ∪
            {shift state' | (symbol state') ∈ state.transitions} ∪
            {accept | (symbol accept) ∈ state.transitions}
  return result
```

3.5. Incremental parse table generation

Now that we have routine *ACTION* checking for initial itemsets, we can easily obtain the incremental parse table generator *IPG* also. *IPG* itself is just *LPG* combined with *ADD*, the routine that modifies an existing parse table when a rule is added to the grammar. The implementation is as follows: *ADD* returns all itemsets affected by the modification in the grammar to their initial state. When the parser needs them, they will be re-completed by *EXPAND* in accordance with the modified grammar.

$ADD(A ::= \beta)$:
  Add $A ::= \beta$ to the Grammar
  for each itemset with a transition on A **do**
    Return the itemset to its state "initial"
  od

Routine *DELETE*, used to delete a rule from a parser, works in a similar way. As said before, the subtler design decisions concerning incremental parser generation are not discussed here; the reader is referred to [HKR89] for a more elaborate explanation.

4. Modular parser generation as a generalization of incremental parser generation

Now that we have described *IPG*, we can continue our presentation of *MPG*. We first explain why combining two parsers is not compatible with the generation techniques used in *IPG*, and then we discuss our solution for modular parser generation in detail. Next, we present the algorithms for *MPG*, and finally we give an example of a modular syntax definition and the parsers generated for it.

4.1. Taking the union of parsers does not work

As stated in section 2.2, the main problem in modular parser generation is combining independently generated parsers. More specifically, how can two graphs of itemsets be combined into the graph corresponding to the union of the associated grammars? It may happen that certain itemsets occur in both graphs, while they should occur only once in the combination, that kernels of certain itemsets are to be merged, and that transitions are to be added from itemsets in one graph to itemsets in the other. Only when the two defining grammars do not interfere in any manner, the resulting graph will consist of two distinct subgraphs. In general, this will not be the case. The decision which modifications to make is very similar to generating a parser for the combination of the grammars.

But even if we would know how to modify and join graphs of itemsets, there still is a problem: how should the required modifications be recorded? If we modify the graphs generated for the sub-grammars directly, we cannot use them separately any more, but if we copy them first, we obtain many graphs of itemsets generated for a single modular syntax definition. This multiplies the time needed if the grammar is modified, as the necessary updates must then be propagated to all graphs affected by the modification.

4.2. Restricted Parsing

We propose an entirely different solution to the problem of taking the union of parsers, which we call *restricted parsing*. We generate one big parser for the union of a given set of modules, but restrict parsing in such a way that only the parts of the parser that correspond to the current module are used. This can be done relatively easily: (1) all rules are tagged with the name of the module in which they are defined, (2) the import graph of the module the parser works for is known. Only rules of modules in that import graph may be used by the parser. Reductions according to rules whose module is not in the import graph are inhibited, and all parsers that try such a reduction should die. This has as effect that only parses consisting entirely of valid rules will succeed. In addition to this, we also disallow transitions to itemsets that do not have rules in their kernel whose module belongs to the current import graph, as we can be sure that parses encountering those itemsets will eventually be forbidden by the restrictions on reductions.

Restricted parsing is a simple and cheap extension of the method used in *IPG*. It has the advantage that the parser can switch to another current module with little overhead, and that all lazy and incremental properties of *IPG* are inherited by the modular system. Removing or adding an import in a module does not take any time at all.

A drawback of the method is that it may happen that part of the parser for a module is invalidated by modifying a module which is not in its import graph. The required recomputation of the graph has then no effect on the behaviour of the parser for the module itself, as all effects will be filtered out again by the restrictions.

If we compare two parsers, one using a parse table specially generated for its source module, and one obtained by the restrictions from a larger parse table, we see that the steps taken by them are identical. This means that our solution does not affect the number of steps taken by a parser. However, the computation of the restrictions does affect the time needed for each step. To avoid this we compute the restrictions only once during parsing, and record the trimmed versions of the reductions and transitions in the itemset.

Next, it may of course be that the union of all modules defines parts of the parser that will never be visited by any parser for a module. As we use a lazy parser generation technique, these parts will then never be generated, however.

The $\Box$-operator of section 2.3 restricts a parser to a sub-grammar. We implement this operator with routine *SPECIALIZE* which restricts the actions prescribed by *ACTION* and *GOTO* according to a set of module names. As rules are tagged with the name of their module, *SPECIALIZE* has the same information as the $\Box$-operator.

4.3. The implementation of *MPG*

The modifications needed to adapt *IPG* to modular parser generation and parsing are:

- Rules are tagged with their module and become of the form $<M{:}A ::= \beta>$, in the dotted case $<M{:}A ::= \alpha\bullet\beta>$. M is the name of the module and the $< >$ brackets are added for readability.
- Before parsing, the module for which is parsed is communicated to the system with the call *PARSE-FOR-MODULE(M)*. This routine stores the set of imported modules of M, which is needed to implement the restrictions on the parser.
- Routine *SPECIALIZE* computes the actions which are valid for the currently usable modules. This information is stored in two new fields of itemsets *specialized-transitions* and *specialized-reductions*. We allow the type of an itemset to be *specialized* also. This new type denotes whether the restrictions have already been computed for the itemset, and makes it possible to compute the restrictions lazily at the first visit of the itemset by the parser.
  SPECIALIZE restricts the allowed actions in the following manner:
  - It only allows reductions according to rules which belong to a module contained in the set of current modules.
  - In order to stop parsers at the earliest possible moment, it only allows shift actions and goto transitions to itemsets which contain at least one rule in their kernel that belongs to the set of current modules. The kernel of an itemset is the set of rules possibly being recognized by the parser; if the kernel does not contain any rule belonging to the set of current modules, each of these possibilities will be forbidden when fully recognized. So we can already forbid such a parse in this stage. The *transition* under START to the general accept itemset with kernel $\{<all{:}\ \mathrm{START}' ::= \mathrm{START}\bullet>\}$ is always allowed.
- *ACTION* now has to check also for itemsets which are not yet specialized, and the actions it returns are derived from the fields *specialized-transitions* and *specialized-reductions*.

To summarize, for a modular grammar definition $\{<N_1, I_1, G_1>, \ldots, <N_n, I_n, G_n>\}$, we generate one (too big) parser using $IPG(G_1 \cup \cdots \cup G_n)$. If we want to parse according to module M_i, we must first call *PARSE-FOR-MODULE*(M_i). This routine computes the modules in the import graph of M_i using:

import-set($<N, \{N_1, \ldots, N_n\}, G>$) =
$\{N\} \cup$ *import-set*($<N_1, I_1, G_1>$) $\cup \cdots \cup$ *import-set*($<N_n, I_n, G_n>$),
and stores this set in a global variable. Next, it makes itemsets that were specialized towards the previous selection of type *complete* again. During parsing *ACTION* will be called. When this routine encounters an itemset which is not yet *specialized*, it will use *SPECIALIZE* to restrict the possible actions according to the set of current modules. Itemsets can now have as type *initial*, *complete* and *specialized*. Fig. 3 shows how the various routines can modify the type of an itemset.

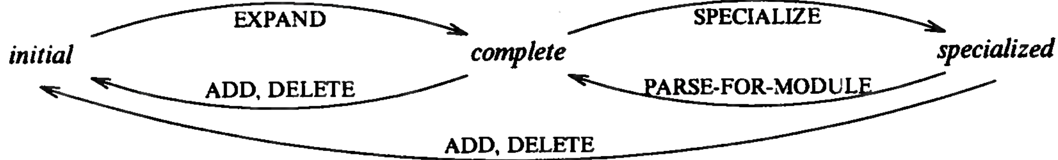


Fig. 3: The routines that modify the *type* of an itemset

4.4. The algorithms for *MPG*

The parser generator of *MPG* is the same as that of *IPG*, and will be called with the union of the grammars of all given modules.

MPG(*Grammar*):
  Grammar := *Grammar* $\cup$ {<all: START′ ::= START>}
  start-itemset := *new*(*itemset*)
  start-itemset.kernel, *start-itemset.type* := {<all: START′ ::= •START>}, initial
  Itemsets := {*start-itemset*}
  return *start-itemset*

Routines *EXPAND* and *K-CLOSURE* now use the new format of rules.

K-CLOSURE(*kernel*):
  Closure := *kernel*
  while $\exists A, B, \alpha, \beta, \gamma, M, M'$: $<M: A ::= \alpha \bullet B\beta> \in$ *Closure* $\wedge$
    $<M': B ::= \gamma> \in$ *Grammar* $\wedge$
    $<M': B ::= \bullet\gamma> \notin$ *Closure* **do**
    Closure := *Closure* $\cup \{<M': B ::= \bullet\gamma>\}$
  od
  return *Closure*

EXPAND(*itemset*):
  Closure := *K-CLOSURE*(*itemset.kernel*)
  itemset.transitions, *itemset.reductions* := $\emptyset$, $\emptyset$
  for $\forall S \in \{S \mid <M: A ::= \alpha \bullet S\beta > \in$ *Closure*$\}$ **do**
    kernel′ := $\{<M: A ::= \alpha S \bullet b> \mid <M: A ::= \alpha \bullet S\beta> \in$ *Closure*$\}$
    if $\exists$*itemset′* $\in$ *Itemsets*: *itemset′.kernel* = *kernel′* **then**
      itemset.transitions := *itemset.transitions* $\cup$ {(*S itemset′*)}
    else
      itemset′ := *new*(*itemset*)
      itemset′.type, *itemset′.kernel* := initial, *kernel′*
      Itemsets := *Itemsets* $\cup$ {*itemset′*}
      itemset.transitions := *itemset.transitions* $\cup$ {(*S itemset′*)}
    fi
  od
  for $\forall <M: A ::= \beta \bullet> \in$ *Closure* **do**
    if *A* = START′ **then**
      itemset.transitions := *itemset.transitions* $\cup$ {($ accept)}
    else

```
      itemset.reductions := itemset.reductions ∪ {<M: A ::= β >}
    fi
  od
  itemset.type := complete
```

ADD has to use the new format as well:

```
ADD(<M: A ::= β >):
  Grammar := Grammar ∪ {<M: A ::= β >}
  for ∀itemset ∈ Itemsets: itemset.type ≠ initial ∧
                          (A itemset′) ∈ itemset.transitions do
    itemset.type := initial
  od
```

Routine *PARSE-FOR-MODULE* stores the set of modules in the import-graph in a global variable *current-modules* and makes all itemsets which where specialized towards the previous module *complete* again.

```
PARSE-FOR-MODULE(M):
  current-modules := IMPORT-SET(M)
  for ∀itemset ∈ Itemsets: itemset.type = specialized do
    itemset.type := complete
  od
```

ACTION now checks for itemsets with type not equal to *specialized*, and uses the fields *specialized-transitions* and *specialized-reductions* to derive its actions from.

```
ACTION(state, symbol):
  if state.type ≠ specialized then
    if state.type = initial then EXPAND(state) fi
    SPECIALIZE(state)
  fi
  result := {reduce A ::= β | A ::= β ∈ state.specialized-reductions} ∪
            {shift state′ | (symbol state′) ∈ state.specialized-transitions} ∪
            {accept | (symbol accept) ∈ state.transitions}
  return result

GOTO(state, symbol):
  return state′: (symbol state′) ∈ state.specialized-transitions

SPECIALIZE(state):
  state.specialized-reductions :=
        {A ::= β | <M: A ::= β> ∈ state.reductions ∧ M ∈ current-modules}
  state.specialized-transitions :=
        {(symbol state′) | (symbol state′) ∈ state.transitions ∧
                           (∃M ∈ current-modules: <M: A ::= α•β> ∈ state′.kernel ∨
                            state′.kernel = {<all: START′ ::= START• >}))}
  state.type := specialized
```

Note that it may now happen that *GOTO* does not return a state, as the needed transition may have been removed by the restrictions. The parsing algorithm should be ready for this situation.

4.5. An example

We use the modular grammar definition of Fig. 4 as an example:

It consists of three modules *Expressions*, *Statements* and *Stacks*, and both *Statements* and *Stacks* import module *Expressions*. Fig. 5.a shows the graph of itemsets generated for the union of the three modules. If we, for instance, want to parse according for module *Expressions*, this graph is restricted, using the method described above, to that of Fig. 5.b. Note that this graph is not the

module *Expressions*:	module *Statements*:	module *Stacks*:
START ::= *Exp*	START ::= *Stm*	START ::= *Stack*
Exp ::= *Id*	*Stm* ::= if *Exp* then *Stm*	*Stack* ::= empty
Exp ::= *Exp* + *Exp*	*Stm* ::= *Id* := *Exp*	*Stack* ::= push *Exp* on *Stack*
		Exp ::= top *Stack*

Fig. 4: A grammar with three modules

same as the graph that would have been generated for module *Expressions* on its own, as the kernels of itemset 0 is too large; this makes no difference for the parser however, which will perform exactly the same actions.

It is interesting that, while the restrictions on *reductions* are a basic feature of *MPG*, in this example reductions are never restricted. The parser just cannot reach itemsets with such restricted reductions, as the extra restrictions on *transitions* prevent this. In general, the parser will only reach itemsets with disallowed reductions, when it already had to be there in order to parse according to allowed rules.

5. Measurements

5.1. Time consumption inside *MPG*

To give an idea of the relative time consumption of the parser itself, the parser generator and routine *SPECIALIZE*, we did some measurements on the grammar of Fig. 6, which is in fact an extended version of the grammar of Fig. 4. In this grammar both module *Stacks* and module *Statements* import module *Expressions*.

module *Expressions*:	module *Statements*:	module *Stacks*:
START ::= *Exp*	START ::= *StmProg*	START ::= *StackProg*
Exp ::= (*Exp*)	START ::= *Stm*	START ::= *Stack*
Exp ::= − *Exp*	*StmProg* ::= begin *Decl* *Stms* end	*StackProg* ::= begin *StackExps* end
Exp ::= *Id*	*Decl* ::= decl *Ids* ;	*StackExps* ::= *StackExp*
Exp ::= *Int*	*Decl* ::=	*StackExps* ::= *StackExps* ; *StackExp*
Exp ::= *Exp* * *Exp*	*Ids* ::= *Id*	*StackExp* ::= *Id* := *Stack*
Exp ::= *Exp* + *Exp*	*Ids* ::= *Ids* , *Id*	*Stack* ::= *Id*
Exp ::= *Exp* − *Exp*	*Stms* ::= *Stm*	*Stack* ::= empty
	Stms ::= *Stms* ; *Stm*	*Stack* ::= pop *Stack*
	Stm ::= *Id* := *Exp*	*Stack* ::= push *Exp* on *Stack*
	Stm ::= begin *Stms* end	*Exp* ::= length *Stack*
	Stm ::= if *BoolExp* then *Stm*	*Exp* ::= top *Stack*
	Stm ::= repeat *Stm* until *BoolExp*	
	Stm ::= while *BoolExp* do *Stm*	
	Stm ::= case *Id* of *CaseStms*	
	CaseStms ::= *CaseStm*	
	CaseStms ::= *CaseStms* ; *CaseStm*	
	CaseStm ::= *Exp* : *Stm*	
	BoolExp ::= not *BoolExp*	
	BoolExp ::= *Exp* < *Exp*	
	BoolExp ::= *Exp* = *Exp*	

Fig. 6: Grammar used in the measurements

We performed the following measurements in succession:

1 Assemble: Assemble all rules in a grammar structure.
2 *Statements*: Parse a sentence (of 110 tokens) according to module *Statements*.
3 *Statements*: Parse the same sentence once more.
4 *Stacks*: Parse a sentence (of 60 tokens) according to module *Stacks*.

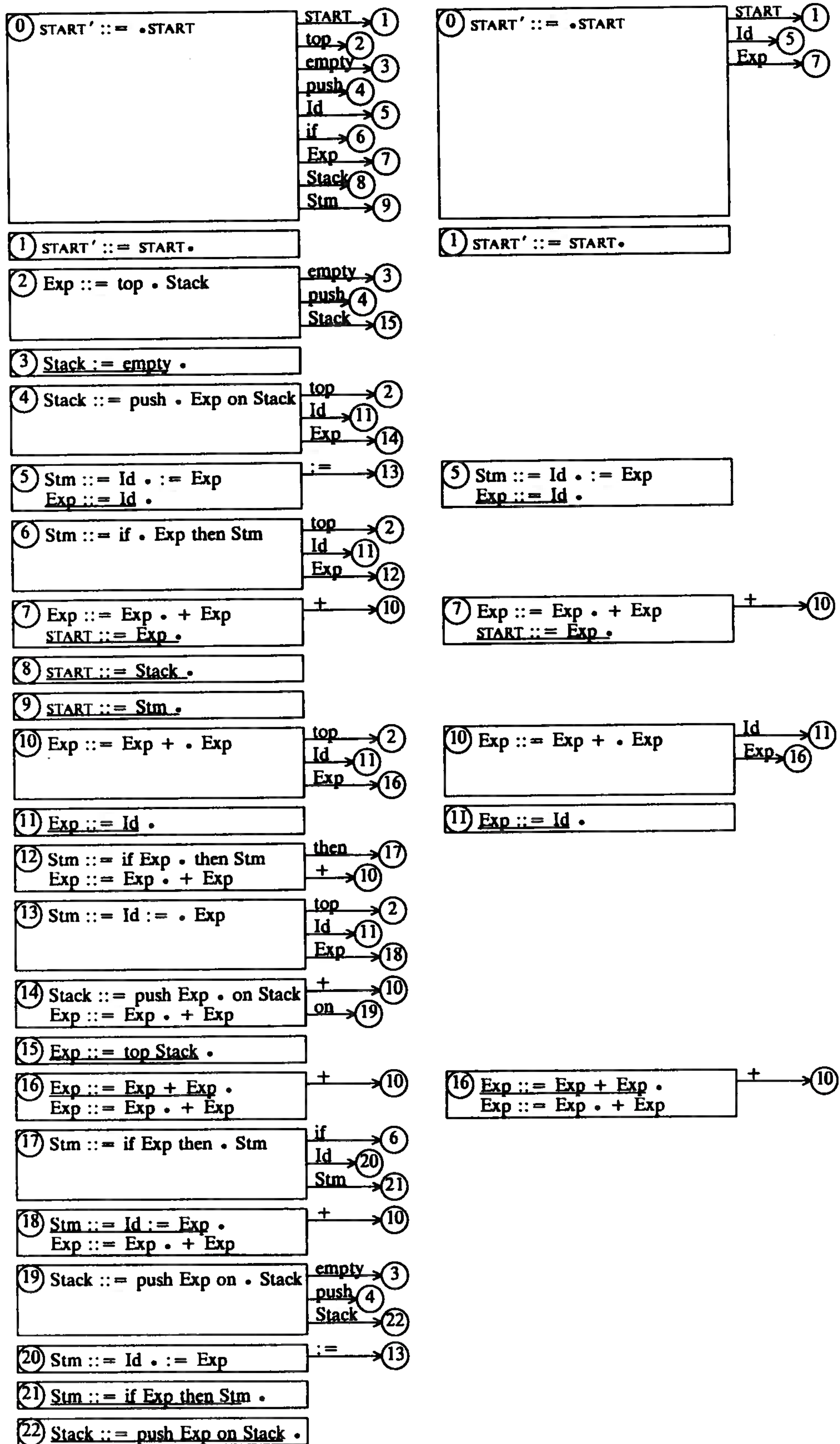


Fig. 5.a: The complete graph, 5.b: The graph restricted to *Expressions*

5 *Stacks*:	Parse the same sentence again.
6 *Statements*:	Re-parse the *Statements* sentence according to module *Statements*.
7 Assemble:	Re-assemble all rules in a new grammar structure.
8 *Stacks*:	Parse the *Stacks* sentence again.

The results of these measurements given in Fig. 7 are the average of 30 repeated executions of them.

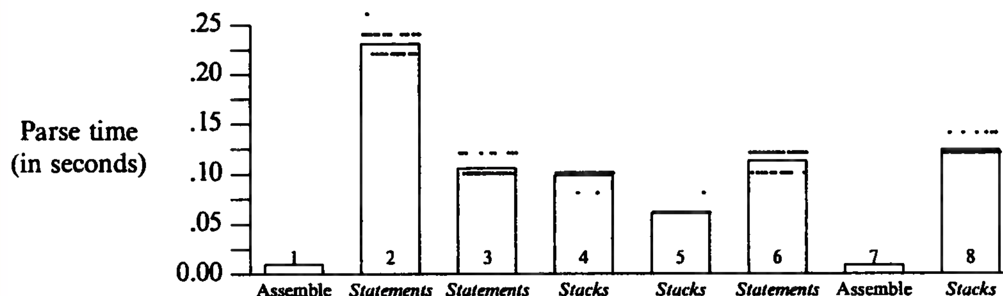


Fig. 7: Eight successive measurements on *MPG*

These results show the following:

- The difference in parse time used between measurement 2 and 3 shows how much time was spend to generate the needed parts of the parser for *Statements*.
- The difference in time consumed between measurement 3 and 6 for *Statements*, shows that re-specializing the already generated parser from *Stacks* to *Statements* takes little time.
- The time needed to parse for *Stacks* differs between measurement 4 and 8. This shows that during the generation for *Statements* in 2, parts of the parser for *Expressions* have been generated, which were of use for *Stacks* also.

5.2. *MPG* versus *IPG*

The generation work performed by *MPG* can always be simulated by just using an ordinary parser generator to generate a separate parser for the full grammar of each module. In this section we will compare the efficiency of *MPG* against such a simulation.

As a test grammar we take the grammar of Pascal, divided into three modules: One module *exp* that describes the syntax of Pascal expressions, one for its statements *stm* and one for complete Pascal programs *prog*. The import relations are that *stm* imports *exp* and *prog* imports *stm* (and thus also *exp*). We want to parse sentences according to each of these modules. When we normalize this modular grammar definition, we obtain for each module an ordinary grammar definition. We use *IPG* to generate a parser for each of these grammars, and *MPG* to generate a parser for the original modular grammar definition.

The measurements we performed are the following: take three input sentences, one for each module, and let them parse in succession by the three *IPG* parsers and by the *MPG* parser specialized towards the appropriate module. These sentences are chosen such that they cover their grammars reasonably well, and large part of the parser has to be generated to parse them. The generation time needed by the two parser generators is shown in Fig. 8.

The generation time used by *IPG* increases with the size of the grammar. In *MPG* however, the generation work done for *exp* is re-used while generating for *stm*, which work is used again in *prog*. *MPG* clearly uses less generation time for *stm* and *prog* as *IPG* needs to do, however the larger generation time of *exp* is the price paid. It is larger because the parser for *exp* is generated by *MPG* in an environment of other rules (those of *stm* and *prog*). These rules are (partly) taken into account while generating the parser for *exp*.

In these measurements we have, to make the comparisons fair, taken *IPG* as non-modular parser generator, as *MPG* and *IPG* only differ in their modular behaviour. In [HKR89] we have

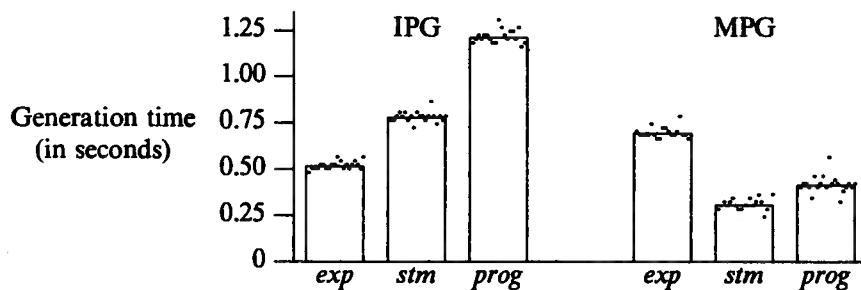


Fig. 8: Generation times of *IPG* and *MPG* on Pascal

compared the efficiency of *IPG* with that of Yacc, and showed that *IPG* generates its parsers about twenty times faster than Yacc does, while the generated parsers are about twice as slow.

6. Conclusion and future work

We have now solved the basic problems of modular parser generation, but modularization does not consist of imports alone. There are some other features that have to be implemented, like hidden sections of modules (i. e. hidden syntax rules may not be used outside the defining module), composition of lexical syntax rules in modules, parameterized modules and imports with renamings.

The modular parser generation technique described in this paper fulfills all requirements given in the introduction. The solution presented (generate one parser for the union of all modules, and restrict that parser according to the set of current modules) results in a separate parser for each module that works as efficient as conventional parsers, while all generation work is re-usable for other modules. The modular parser generator *MPG* is built on top of *IPG*, re-using all implementation work, giving that *MPG* is as flexible as *IPG*.

Acknowledgments

I would like to thank Paul Klint. He entered my room one morning with the remark: "All these troubles you are having with taking the union of two parsers, why don't you just generate the parser for that union, but restrict the reductions of it to allowed rules?". This method, with some fine-tuning, did the trick. Paul Klint himself applied the idea to a modular generator of lexical scanners [Kli89]. Next, I am very grateful to Jan Heering for his careful proofreading of this paper.

References

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers. Principles, Techniques and Tools*, Addison-Wesley (1986).

[FGJM85] K. Futatsugi, J.A. Goguen, J.-P. Jouannaud, and J. Meseguer, "Principles of OBJ2," pp. 52-66 in *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, ACM (1985).

[HKR89] J. Heering, P. Klint, and J. Rekers, "Incremental generation of parsers," pp. 179-191 in *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation*, SIGPLAN Notices **24**(7), ACM Press (1989).

[Kli89] P. Klint, "Scanner generation for modular regular grammars," pp. 291-305 in *Liber Amicorum, J.W. de Bakker, 25 jaar Semantiek*, Centre for Mathematics and Computer Science, Amsterdam (1989).

[Tom85] M. Tomita, *Efficient Parsing for Natural Languages*, Kluwer Academic Publishers (1985).

[Voi86] F. Voisin, "CIGALE: a tool for interactive grammar construction and expression parsing," *Science of Computer Programming* **7**, pp. 61-86 (1986).